C++学习笔记4

2022年3月19日 20:48

- 类
  - 定义：在常规上，我们可以利用struct/class定义类。
  - 在struct里面，里面的内容我们称之为成员函数
    - 成员函数
      - 包括简单的类型
        int a；int b；
      - 也可以包括其他的类
        string a；
      - 也可以包括函数
        - 注意
          - 函数包括声明和定义，其中声明必须要在struct内部，而函数的定义可以在里面也可以在外面
            可以只包括string fun（）；
            也可以包括string fun（）{return 0；}《------包括函数的具体内容
          - 定义于类内部的函数是隐式的inline函数
  - 调用/定义
    - 对于在主函数里面调用成员函数，我们可以使用点运算符进行调用，我们可以利用点运算符调用所有的类里面的函数
      p.a//p.isbn()
    - this与调入
      - 调用的时候的细节/this的引入
        - 在调用的时候，p的地址此时给了一个隐式的指针形参this（在类中成员函数内部任何调用函数的时候都存在），指向p，程序才明白此时是将p的数据进行传递然后运算，才完成了一个跨作用域的调入
        - this是不可以被当作正常的变量的，因为他是一个指向非常量对象的常量指针
      - 调用函数时候的细节
        - 对于类内部的函数，我们有的时候可以把它写为（在定义的时候也要一样）
          int f（int a，int b）const；
        - 对于结尾加const的意思是，将this指针声明为const类型，从而防止更改
          - 如果把指针声明成为this类型的话，那么对于被调用的函数来说，被调用的函数可以读取对象中的内容，但是不能向对象中写入新的值

```
#include <iostream>
#include <string>
using namespace std;
struct a{
    string x;
    int f(int a,int b)const;

};
int
{
    x[2]='s';
    return 0;
}
```

表达式必须是可修改的左值 C/C++(137)

查看问题 快速修复...(Ctrl+.)

- ◇ 常量对象，以及常量对象的引用或指针都只能调用常量成员函数
  - C++里，无权更改本类数据成员的成员函数称为常量函数（加const的）

- 定义成员函数
  - 对于在类外定义成员函数，需要加上作用域（：：）表明其身份，且指明成员再进行定义。
    在类内有声明，类外才能 作用域：：成员（定义。。）
  - 作用域的声明一般在返回类型之后，函数名字之前，作用域的名字就是类的名字

```
int a:: f(int a,int b)
{
    x[2]='s';
    return 0;
}
```

  - 成员函数也可以重载，遵循正常的函数重载规则
  - 可变数据成员
    - 有时，我们希望能修改类的某个数据成员，即使是一个在const成员函数里面，可以通过在变量的声明中加入mutable关键字做到

```
class Screen {
public:
    void some_member() const;
private:
    mutable size_t access_ctr;    // 即使在一个 const 对象内也能被修改
    // 其他成员与之前的版本一致
};
void Screen::some_member() const
{
    ++access_ctr;        // 保存一个计数值，用于记录成员函数被调用的次数
    // 该成员需要完成的其他工作
}
```

- 更改成员函数
  - 对于通过类函数来更改对象，是需要使用对象的指针/引用的，直接修改是不

可以的

```cpp
#include <iostream>
#include <string>
using namespace std;
struct a{
    string x;
    a& f(int a,int b);

};

a& a:: f(int a,int b)
{
    x[2]='s';
    return *this;
}
```

- 非成员函数
  - 非成员函数的定义
    - 类的作者常常需要定义一些辅助函数，比如add，read，print等等，尽管这些函数定义的操作从概念上来说属于类的接口的组成部分，但它们实际上并不属于类本身
    - 我们定义非成员函数的方式与定义其他函数一样，通常把函数的声明和定义分离开来。
    - 如果函数在概念上属于类但不是定义在类中，则它一般应与类声明在同一个头文件内，在这种方式下，用户使用接口的任何部分都只需要引入一个头文件。
    - 一般来说，如果非成员函数是类接口的组成部分，则这些函数声明应该与类在同一个头文件里面
  - 成员函数与非成员函数之间的区别
    - 成员函数的声明包含在类中，在外部（额外的cpp或者主函数后面）需要记得把作用域带上，可以直接调用类内的内容
    - 非成员函数和普通函数一样，声明一般可以放在成员函数的内部，如果需要调用类内的函数等需要使用点运算符，而不能使用this，因为this只能在成员函数中使用
  - io流的特殊调用
    - 对于io流的调用，返回类型便是io类型，传入的时候记得写io类型（cin。。）
      istream&f(istream&is,int i)
      is>>,,,
      ostream&f(ostream&os,int i)
      os<<...
    - io类属于不能被拷贝的类型，因此我们只能通过引用来传递它们

        - 读取和写入操作会改变流的内容，所以两个函数接受的都是普通引用，而非对常量的引用
- 构造函数（初始化）
    - 定义：每个类分别定义了它的对象被初始化的方式，类通过一个或几个特殊的成员函数控制其对象的初始化过程，这些函数叫构造函数
    - 注意
        - 构造函数的任务是初始化类对象的数据成员，无论何时只要类的对象被创建，就会执行构造函数，当没有创建构造函数的时候，编译器会自己生成一个构造函数
            - 当a被创建的时候，就执行了构造函数

```
#include <iostream>
#include <string>
using namespace std;
struct person
{

    person()=default;
    person(const string &s):name(s){}
    string name;
    string address;

};
istream& read(istream& is,person& x);
ostream& print(ostream& os,person& x);
int main()
{
    person a("ssss");
```

        - 构造函数的名字与类名相同，但是没有返回类型，但是也可能有一个可能为空的参数列表和可能为空函数体

          对于一般：person(const string &s):name(s){}

          对于io类：会有个return
        - 不同于其他成员函数，构造函数不能被声明为const的，但是可以将括号内的变量声明为const的
            - 因为当我们创建一个const对象的时候，直到构造函数完成初始化过程，对象才能真正取得其“常量”属性，因此，构造函数在const对象的构造过程中可以向其写值

```
{

    person()=default;
    person(const string &s):name(s){}
```

        - 类可以包括多个构造函数，和其他的重载函数差不多，不同的构造函数之间必须在参数数量上有所区别
            - 这里也包括了和重载函数一样，如果构造函数后，传入的数字必须与空格对应，否则会出现错误
        - 为什么某些类不能依赖于合成的默认构造函数
            - 只有当类没有声明任何构造函数时，编译器才会自动地生成默认构造函

数
- 如果类包含有内置类型或者复合类型的成员，则只有当这些成员全部被赋予了类内初始值的时候，这个类才适合于使用合成的默认构造函数
- 有的时候编译器不能为某些类合成默认的构造函数

- 通常情况下，构造函数使用的类内初始值不失为一种好的选择，因为只要这样的初始值存在我们就能确保为成员赋予了一个正确的值。类内初始值必须要进行赋值，而不能使用括号等直接初始化。
- 构造函数不应该轻易的覆盖掉类内初始值，除非新赋的值与原值不同。如果你不能使用类内初始值，则所有构造函数都应该显式地初始化每个内置类型的成员

- 构造函数的形式
  - person（）=default
    - 我们希望这个函数的作用完全等同于之前使用的合成默认构造函数
    - 如果=default在类的内部，则是内联函数，如果=default在外部，则不是内联函数
  - person（string a）：name（s）{}
    - 这个时候，在初始化的时候只会按照要求初始化name，如果有其他成分，则其他成分会被默认初始化
    - 但是如果在运用的时候，注意不能多不能少
      错误：person a（“sss”，“sss”）；
- 在类的外面构造函数
  - 基本与类内一致，但是与其他几个构造函数不同，以istream为参数且在类内的构造函数需要执行一些实际的操作，也需要指定作用域和成员
    person：：f（istream& is）：{read（is，*this）；}

○ 再探构造函数
- 构造函数初始值列表
  - 构造函数的初始值有时必不可少
    - 如果成员是const，引用或者属于某种未提供默认构造函数的类类型，我们必须通过构造函数初始值列表为这些成员提供初值
  - 成员初始化的顺序
    - 第一个成员先被初始化，然后第二个，以此推类
    - 最好令构造函数初始值的顺序与成员声明的顺序保持一致，而且如果可能的话，尽量避免使用某些成员初始化其他成员
    - 如果可能的话，最好使用构造函数的参数作为成员的初始值，而尽量避免使用同一个对象的其他成员。
      - 好处是我们可以不必考虑成员初始化的顺序
    - 如果一个构造函数为所有参数都提供了默认实参，则它实际上也定义了默认构造函数
  - 委托构造函数
    - 一个委托构造函数使它所属类的其他构造函数执行他自己的初始化流程

```
class Sales_data {
public:
    // 非委托构造函数使用对应的实参初始化成员
    Sales_data(std::string s, unsigned cnt, double price):
            bookNo(s), units_sold(cnt), revenue(cnt*price) { }
    // 其余构造函数全都委托给另一个构造函数
    Sales_data(): Sales_data("", 0, 0) {}
    Sales_data(std::string s): Sales_data(s, 0,0) {}
    Sales_data(std::istream &is): Sales_data()
                                    { read(is, *this); }
    // 其他成员与之前的版本一致
};
```

- 总结：默认构造函数的作用
  - 默认初始化发生
    - 当我们在块作用域内不使用任何初始值定义一个非静态变量或数组时
    - 当一个类本身含有类类型的成员且使用合成的默认构造函数时
    - 当类类型的成员没有在构造函数初始值列表中显示初始化时
  - 值初始化
    - 在数组初始化的过程中如果我们提供的初始值数量少于数组的大小时
    - 当我们不使用初始值定义一个局部静态变量时

- 作用域
  - 类也一样，利用大括号表示了作用域，其符合作用域规则
- 类的访问控制，与封装
  - 定义在public的说明符说明成员在整个程序内可被访问，public成员定义类的接口
  - 定义在private说明符之后的成员可以被类的成员函数访问，但是不能使用该类的代码访问，private部分封装了类的实现细节

    class xxx{

    public:

    xxxx;

    private:

    xxx;
  - 使用class和struct定义类唯一的区别就是默认的访问权限
  - 在类的定义中，可以包含0个或者多个访问说明符，并且对于某个访问说明符能出现多少次以及能出现在哪里都没有严格的规定。每个访问说明符指定下来的成员访问级别，有效范围直到出现下一个访问说明符或者到达类的结尾为止
  - 封装
    - 定义：封装是指保护类的成员不被随意访问的能力。通过把类的实现细节设置为private，我们就能完成类的封装。封装实现了类的接口和实现的分离。
    - 优点：
      - 确保用户代码不会无意间破坏封装对象的状态。
      - 被封装的类具体实现细节可以随时改变，而无需调整用户级别的代码
      - 防止由于用户原因造成数据被破坏。
    - 注意

        - 一般来说，作为接口的一部分，构造函数和一部分成员函数应该定义在public说明符之后，而数据成员和作为实现部分的函数则应该跟在private说明符之后
- 友元函数
    - 类允许其他类或者函数访问它的非共有成员，方法是令其他类或者函数成为它的友元
    - 如果类想把一个函数作为它的友元，只需要增加一条以friend关键字开始的函数声明语句即可
        -
```
class person
{
    friend istream& read(istream& is,person& x);
    friend ostream& print(ostream& os,person& x);
```

```
istream& read(istream& is,person& x);
ostream& print(ostream& os,person& x);
```

    - 。我们通常把类的友元函数的声明放在类内这个位置
        -
```
class person
{

    friend istream& read(istream& is,person& x);
    friend ostream& print(ostream& os,person& x);
public:
    person()=default;
    person(const string &s):name(s){}
    person(const string &s,const string &c):name(s),address(c){}
    int f(int a,int b,person x);
private:
    string name="sss";
    string address="sss";
    int f2(int a,int b);



};
```

    - 声明友元函数之后在外面还要再写普通声明的！！
    - 友元关系不存在传递性，对于一个是A类友元类的B，B的友元函数是无法直接访问A的私有类的（A可以访问B，但是A不可以通过B的友元访问C）
- 使用类型别名定义函数
    - 除了定义数据和函数成员外，类还可以自定义某种类型在类中的别名。由类定义的类型名字和其他成员一样存在访问限制。
    - 用来定义类型的成员必须先定义后使用
- 类数据成员的初始值
    - 可能会有多个相同的类进行初始化，在C++11标准中，最好的方式就是把这个默认值声明成为一个类内初始值
    - 当我们提供一个类内初始值时候，必须以等于号或者花括号表示
- 类类型，类的声明与定义
    - 类类型
        - 即使两个类的成员相同，但名字不同，则类不同，类的区分为名字，名字相同类才能相等

        - 对于一个类来说，它的成员和其他任何类（或者说任何其他作用域）的成员都不是一回事
    - 我们可以把类名作为类型的名字使用，从而直接指向类类型。
  - 类的声明
    - 就像可以把函数声明和定义分开一样，我们也能仅仅声明类而暂时不定义它，此时我们已知声明的类是一个类，但是不知道它包含哪些内容
    - 在他声明之后定义之前是一种不完全类型，不完全类型只能在非常有限的情景下使用
      - 可以定义指向这种类型的指针或者是引用
      - 可以声明（但是不能定义）以不完全类型作为参数或者返回类型的函数
- 类的作用域
  - 每个类都会定义它自己的作用域。
  - 在类的作用域之外，普通的数据和函数成员只能由对象，引用，或者指针使用成员访问运算符，对于类类型成员则使用作用域运算符访问
    - 不论哪种情况，跟在运算符之后的名字都必须是对应类的成员
  - 名字查找与类的作用域
    - 名字查找
      - 首先，在名字所在的块中寻找声明语句，只考虑在名字的使用之前出现的声明
      - 如果没找到，继续查找外层作用域
      - 如果最终没有找到匹配声明，则程序报错
    - 类的定义
      - 首先编译成员的声明
      - 直到类全部可见后才编译函数体
    - 用于类成员声明的名字查找
      - 这种两阶段的处理方式只适用于成员函数中使用 的名字
      - 声明中使用的名字，包括返回类型或者参数列表中使用的名字，都必须在使用前确保可见。
      - 如果某个成员的声明中使用了类中尚未出现的名字，则编译器将会在定义该类的作用域中继续查找
    - 类型名字的特殊说明
      - 一般来说，内层作用域可以重新定义外层作用域中的名字，即使该名字已经在内层作用域中使用过。
      - 但是在类中，如果成员使用了外层作用域中某个名字，而该名字代表一种类型，则类不能在之后重新定义该名字
    - 成员定义中的普通块作用域的名字查找
      - 首先，在成员函数内查找该名字的声明。和前面一样，只有在函数使用之前出现的声明才被考虑
      - 如果在成员函数内没有找到，则在类内继续查找，这时类的所有成员都可以被考虑

        - 如果类内也没有找到该名字的声明，在成员函数定义之前的作用域内继续查找
        - 简而言之，由小到大：成员函数---》类的作用域---》全域
    - 注意
        - 编译器处理完类中的全部声明才会处理成员函数的定义
        - 类型名的定义通常出现在类的开始处，这样就能确保所有使用该类型的成员都出现在类名的定义之后
        - 对于因为部分外层声明的对象在内层再次声明，如果需要调用外层的对象可以使用作用域进行声明
- 隐式的类类型转换
    - 如果构造函数只接受一个实参，则它实际上定义了转换为此类类型的隐式转换机制，有时候我们把这种构造函数称作转换构造函数
        - 1.将一个string s转换为 sata类的一个成员
        - 2.可以直接将这个string类型的成员隐式的转换为sata对象传入sata对象才能传入的参数
    - 类类型的转换只允许一步，如果需要第二次需要强制的显性转换
    - 抑制构造函数定义的隐式转换（explicit）
        - 我们可以通过将构造函数声明为explicit加以阻止
            - ```
              class Sales_data {
              public:
                  Sales_data() = default;
                  Sales_data(const std::string &s, unsigned n, double p):
                             bookNo(s), units_sold(n), revenue(p*n) { }
                  explicit Sales_data(const std::string &s): bookNo(s) { }
                  explicit Sales_data(std::istream&);
                  // 其他成员与之前的版本一致
              };
              ```
        - 注意
            - 关键词explicit只对一个实参构造函数有效。需要多个实参的构造函数不能用于执行隐式转换
            - 只能在类内声明狗道函数时候使用explicit关键字，在类外部定义时候不必重复
            - 我们只能使用直接初始化而不能使用explicit构造函数
                - 当我们使用exolicit关键字构造函数的时候，它将只能以直接初始化的形式使用，而且编译器将不会在自动转换过程中使用该构造函数
            - 为转换显示地使用构造函数
                - 尽管编译器不会将exolicit的构造函数用于隐式的转换过程，但是我们可以进行强制类型转换
                - ```
                  // 正确：实参是一个显式构造的 Sales_data 对象
                  item.combine(Sales_data(null_book));
                  // 正确：static_cast 可以使用 explicit 的构造函数
                  item.combine(static_cast<Sales_data>(cin));
                  ```
                -
- 聚合类

- 聚合类使得用户可以直接访问其成员，并且具有特殊的初始化语法形式
- 当一个类满足如下条件时，我们说他是聚合的
  - 所有成员都是public的
  - 没有定义任何构造函数
  - 没有类内初始值
  - 没有基类，也没有virtual函数。
- 初始化
  - 我们可以提供一个花括号括起来的成员初始值列表，并用它初始化聚合类数据成员

    struct data{
     int a;
     int b;
    }
  - 初始化的顺序必须与声明的顺序一致，也就是说，第一个成员的初始值要放在第一个，然后是第二个，以此推类

    data a,b;
    a.a=2,a.b=3;
    data b={2,3};
  - 与初始化数组元素的规则一样，如果初始值列表中的元素个数少于类的成员数量，则靠后的成员被值初始化。初始值列表的元素个数绝对不能超过类的成员数量
- 缺点
  - 要求类的所有成员都是public的
  - 将正确初始化每个对象的每个成员的重任交给了类的用户（而非类的作者）。因为用户很容易忘掉某个初始值，或者提供一个不恰当的初始值，所以这样的初始化过程冗长乏味且容易出错
  - 添加或删除一个成员后，所有的初始化语句都需要更新

○ 字面值常量类
- 数据成员都是字面值的类型的聚合类是字面值常量类
- 如果一个类不是聚合类，但是它符合下述要求，则他也是一个字面值常量类
  - 数据成员都必须是字面值类型
  - 类必须至少含有一个constexpr构造函数
  - 如果一个数据成员含有类的初始值，则内置类型成员初始值必须是一条常量表达式
  - 或者如果成员属于某种类类型，则初始值必须使用成员自己的constexpr函数
  - 类必须使用析构函数的默认定义，该成员负责销毁类对象
- constexpr构造函数可以声明为=default的形式（或者是删除函数的形式）
- 否则，constexpr构造函数就必须既符合构造函数的要求（意味着它能拥有的唯一可执行语句就是返回语句）

- 类的静态成员（static）
  - 我们通过在成员的声明之前加上关键词static使得其与类关联在一起。和其他成员一样，静态成员可以是public或者private的。静态数据成员的类型可以是常量/引用/指针/类类型等
  - 类的静态成员存在于任何对象之外，对象中不包含任何与静态数据成员有关的数据
  - 类似的，静态成员函数也不与任何对象绑定在一起，他们不包含this指针。作为结果，静态成员函数不能声明为const的，而且我们也不能在static函数体内使用指针this，这一限制既适用于this的显式使用，也对调用非静态成员的隐式使用有效
  - 虽然静态成员不属于类的某个对象，但是对于非成员函数我们仍然可以使用类的对象，引用或者指针来访问静态成员，对于成员函数不用通过作用域运算符就能直接访问那个成员
  - 定义静态成员
    - 和其他成员函数一样，我们既可以在类的内部也可以在类的外部定义静态成员函数
    - 当在类的外部定义静态成员的时候，不能重复static关键词，该关键词只出现在类的内部
    - 和类的所有成员一样，当我们指向类外部的静态成员时，必须指明成员所属的类名。static关键词则只出现在类内部的声明语句中
    - 因为静态数据成员不属于类的任何一个对象，所以他们并不是在创建类的对象的时候被定义的。这意味着它们不是由类的构造函数初始化的
    - 我们不能在类内部初始化静态成员。相反的，必须在类的外部定义和初始化每个静态成员。和其他成员一样，一个静态数据成员只能定义一次
      - 类型名+作用域+函数名=xxx
    - 要想确保对象只定义了一次，最好的办法就是把静态成员的定义与其他非内联函数的定义放在同一个文件中
  - 静态成员的类内初始化
    - 通常情况下，类的静态成员不应该在类内部初始化。然而，我们可以为静态成员提供const整数类型的初始值，不过要求静态成员必须是字面值constexpr。
    - 初始值必须式常量表达式，因为这些成员本身就是常量表达式所以他们能用在适合于常量表达式的地方
    - 即使一个常量静态数据成员在类内部被初始化了，通常情况下也应该在类的外部定义一个该成员
  - 静态成员能用于某些场景而普通成员不能
    - 静态数据成员类型可以就是他所属的类类型，而非静态成员则收到限制，只能声明成它所属类的指针或引用
      -
```
class Bar {
public:
    // ...
private:
    static Bar mem1;              // 正确：静态成员可以是不完全类型
    Bar *mem2;                    // 正确：指针成员可以是不完全类型
    Bar mem3;                     // 错误：数据成员必须是完全类型
};
```

        - 我们可以使用静态成员作为默认实参，非静态数据成员不能作为默认实参，因为值本身属于对象的一部分
    - 优点
        - 静态成员的优点包括：作用域位于类的范围之内，避免与其他类的成员或者全局作用域的名字冲突；可以是私有成员，而全局对象不可以；通过阅读程序可以非常容易地看出静态成员与特定类关联，使得程序的含义清晰明了。
        - 静态成员与普通成员的区别主要体现在普通成员与类的对象关联，是某个具体对象的组成部分；而静态成员不从属于任何具体的对象，它由该类的所有对象共享。另外，还有一个细微的区别，静态成员可以作为默认实参，而普通数据成员不能作为默认实参。
- 其他概念
    - 不同的编程角色
        - 程序员们常把运行其程序的人称作用户（user）。类似的，类的设计者也是为其用户设计并实现一个类的人，显然，类的用户是程序员，而非应用程序的最终使用者
    - 尽管当类的定义发生改变时无需更改用户代码，但是使用了该类的源文件必须重新编译
    - 类的接口，成员函数（inline/外联）都应该与相应的类定义在同一个头文件中
- 注意
    - 注意点运算符的作用
        - 当我们在使用点运算符后，注意点运算符所指的对象（this）已经被传入，调用的对象进入共有区域
        - 当调用到成员函数的时候，在成员函数运算的操作的时候可以直接调用他的私有成员，当调用它的成员的时候，this也是已经被传入的东西
        - 对于友元函数，当我们使用friend后也只是可以利用点函数读取到内部的所有内容，包括私有部分内容（并没有传入对象）
        - 可以读取到私有部分的函数
            - 成员函数，可以直接访问，因为已经通过this传入对象
            - 友元函数，可以通过。运算符进行访问，因为没有通过this传入对象
        - 在类的外部，可以定义共有也可以定义私有函数
        - 对于普通的函数是无法调用私有成分的
    - 只有在作用域内部，才能读取作用域内部的内容（包括公有或者私有）。在作用域外部，使用点函数只能访问到作用域公共部分的内容，除非把他声明为友元函数才能访问到它的私有部分
    - 对于定义，只要是任何在里面的东西，都要带上作用域
    - 对于有输入输出的定义
        - 传入的参数必须都要是引用，如果是is/ostream开头记得返回is/os
        - 对于构造函数，如果有需要使用this指针指定传入的内容

```
Screen(istream& is){is>>this->name;}
```

    - 对于公共代码使用私有功能函数
        - 一个基本的愿望是避免在多处使用同样的代码
        - 我们预期随着类的发展，display函数有可能变得更加复杂，此时把相应的操作写在一处而非两处的作用就比较明显了

        - 我们可能在开发过程中给私有功能函数添加某些调试信息，而这些信息将在代码最终产品版本中去掉，在私有功能函数一处添加或删除这些信息就要更容易一些
        - 这个额外的函数调用不会增加任何的开销，因为我们在类的内部定义了函数，所以他隐式的被声明为了内联函数，这样的话，调用函数就不会带来任何额外的运行时开销
        - 在实践中，设计良好 的C++代码常常包含大量类似于私有功能函数的函数，通过调用这些函数，可以完成一组其他函数的“实际”工作
    - 如果一个构造函数为所有参数都提供了默认实参，则它实际上也定义了默认实参
- 总结
    - 类有两项基本能力
        - 一是数据抽象能力，即定义数据成员和函数成员的能力
        - 二是封装，即保护类的成员不被随意访问的能力
            - 通过将类的细节实现为private，我们就能完成类的封装。
            - 类可以将其他类或者函数设为友元，这样就能访问类的非共有成员了
    - 类可以定义一种特殊的成员函数：构造函数
        - 其作用是可以控制初始化对象的方式
        - 构造函数可以重载，构造函数应该使用构造函数初始化值的列表来初始化所有的成员
    - 类还能定义静态成员
        - 一个可变的成员永远都不会是const，即使在const成员函数内部也能修改她的值，一个静态成员可以是函数也可以是数据，静态成员存在于所有对象之外